User Manual

for

Sprachinspektor SDK

A C/C++ software development kit to identify language and character encoding

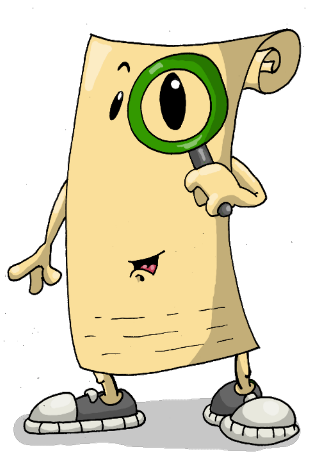
Covers version 4.0.0

Lingua Systems
NATURAL
LANGUAGE
PROCESSING

Sprachinspektor SDK User Manual, published April 11, 2014.

Lingua-Systems Software GmbH, Gerichtsstraße 42, 44649 Herne, Germany, *info@lingua-systems.com*

If you have problems using the links or get aware of any faults, feel free to give a brief hint on it via *support@lingua-systems.com*.



Contents

About this manual

This manual addresses users with experience in C/C++ programming and at least a basic knowledge of library usage.

The manual provides a short introduction to the library, its supported languages and character encodings as well as instructions how to install the Sprachinspektor software package. Afterwards the complete API is introduced along with the possibilities of error handling. A complete usage example is attached in appendix A.

For a quickstart have a look at the function reference in the documentation of the application programming interface (chapter 4.3 on page 12).

Administrators who want to install the software get all necessary information in chapter 3, page 9.

Conventions used in this Manual

At several points of this manual it is necessary to make a distinction between strings that may not contain embedded *NUL* characters and those which may – due to their special charset – potentially contain *NUL* characters. In this manual, the former are called "*Strings*" while the latter are called "*Byte Strings*".

1. Introduction

Sprachinspektor SDK provides a shared C/C++ library that identifies the language and character encoding of textual input. The input can either be a file or in various string formats.

The library computes the results fast and reliable and has no software dependencies except for the standard C and thread library. Therefore Sprachinspektor SDK can be easily integrated on all supported platforms and works efficiently even on dated hardware.

This version of the Sprachinspektor SDK library supports 29 languages and 39 encodings. Additionally Sprachinspektor SDK identifies 10 languages as well if they have been transliterated according to one of the supported standards.

2. Supported Languages and Character Encodings

Currently, 29 languages are supported. The supported 39 encodings cover commonly used encodings as well as deprecated ones.

Language	ISO 639-3 Code	Character Encoding
Arabic	ara	UTF-32, UTF-16, UTF-8, ISO-8859-6, Windows-1256, MacArabic, CP 720
Bokmål (Norwegian)	nob	UTF-32, UTF-16, UTF-8, ISO-8859-1, Windows-1252, MacRoman, CP 850, ASCII
Bulgarian	bul	UTF-32, UTF-16, UTF-8, ISO-8859-5, Windows-1251, MacCyrillic, CP 855, CP 866, KOI8-R
Czech	ces	UTF-32, UTF-16, UTF-8, ISO-8859-2, Windows-1250, MacCentralEurope, CP 852
Danish	dan	UTF-32, UTF-16, UTF-8, ISO-8859-1, Windows-1252, MacRoman, CP 850, ASCII
Dutch	nld	UTF-32, UTF-16, UTF-8, ISO-8859-1, ISO-8859-15, Windows-1252, MacRoman, CP 850, ASCII
English	eng	UTF-32, UTF-16, UTF-8, ISO-8859-1, Windows-1252, MacRoman, CP 850, ASCII
Estonian	est	UTF-32, UTF-16, UTF-8, ISO-8859-4, Windows-1257, MacCentralEurope, CP 775, ASCII
Finnish	fin	UTF-32, UTF-16, UTF-8, ISO-8859-1, ISO-8859-15, Windows-1252, MacRoman, CP 850, ASCII
French	fra	UTF-32, UTF-16, UTF-8, ISO-8859-1, ISO-8859-15, Windows-1252, MacRoman, CP 850, ASCII
German	deu	UTF-32, UTF-16, UTF-8, ISO-8859-1, ISO-8859-15, Windows-1252, MacRoman, CP 850, ASCII
Greek	ell	UTF-32, UTF-16, UTF-8, ISO-8859-7, Windows-1253, MacGreek, CP 737
Hungarian	hun	UTF-32, UTF-16, UTF-8, ISO-8859-2, ISO-8859-16, Windows-1250, MacCentralEurope, CP 852
Irish (Gaelic)	gle	UTF-32, UTF-16, UTF-8, ISO-8859-1, Windows-1252, MacRoman, CP 850, ASCII
Italian	ita	UTF-32, UTF-16, UTF-8, ISO-8859-1, ISO-8859-16, Windows-1252, MacRoman, CP 850, ASCII
Lithuanian	lit	UTF-32, UTF-16, UTF-8, ISO-8859-4, Windows-1257, MacCentralEurope, CP 775, ASCII

Language	ISO 639-3 Code	Character Encoding
Latvian	lav	UTF-32, UTF-16, UTF-8, ISO-8859-4, Windows-1257, MacCentralEurope, CP 775, ASCII
Maltese	mlt	UTF-32, UTF-16, UTF-8, ISO-8859-3
Mandarin (Chinese)	cmn	UTF-32, UTF-16, UTF-8, Big5, GB2312
Nynorsk (Norwegian)	nno	UTF-32, UTF-16, UTF-8, ISO-8859-1, Windows-1252, MacRoman, CP 850, ASCII
Polish	pol	UTF-32, UTF-16, UTF-8, ISO-8859-2, ISO-8859-16, Windows-1250, MacCentralEurope, CP 852
Portuguese	por	UTF-32, UTF-16, UTF-8, ISO-8859-1, ISO-8859-15, Windows-1252, MacRoman, CP 850, ASCII
Romanian	ron	UTF-32, UTF-16, UTF-8, ISO-8859-2, Windows-1250, MacRomanian, CP 852
Russian	rus	UTF-32, UTF-16, UTF-8, ISO-8859-5, Windows-1251, MacCyrillic, CP 855, CP 866, KOI8-R
Swedish	swe	UTF-32, UTF-16, UTF-8, ISO-8859-1, Windows-1252, MacRoman, CP 850, ASCII
Slovak	slk	UTF-32, UTF-16, UTF-8, ISO-8859-2, Windows-1250, MacCentralEurope, CP 852
Slovenian	slv	UTF-32, UTF-16, UTF-8, ISO-8859-2, ISO-8859-16, Windows-1250, MacCentralEurope, CP 852, ASCII
Spanish	spa	UTF-32, UTF-16, UTF-8, ISO-8859-1, ISO-8859-15, Windows-1252, MacRoman, CP 850, ASCII
Ukrainian	ukr	UTF-32, UTF-16, UTF-8, Windows-1251, MacUkrainian, KOI8-U

In addition 10 languages can be identified even in transliterated forms. The 12 supported transliterations cover official norms and commonly used transliterations as found in emails.

Language	Transliteration	Character Encodings
Bulgarian	ISO 9	UTF-32, UTF-16, UTF-8, ASCII
	DIN 1460	UTF-32, UTF-16, UTF-8, ASCII, Windows-1250
	Streamlined System	UTF-32, UTF-16, UTF-8, ASCII, Windows-1250
Czech	common	UTF-32, UTF-16, UTF-8, ASCII, ISO-8859-1
German	common	UTF-32, UTF-16, UTF-8, ASCII, ISO-8859-1
Greek	ISO 843	UTF-32, UTF-16, UTF-8, ASCII
	DIN 31634	UTF-32, UTF-16, UTF-8, ASCII
	Greeklish	UTF-32, UTF-16, UTF-8, ASCII, ISO-8859-1
Polish	common	UTF-32, UTF-16, UTF-8, ASCII, ISO-8859-1
Romanian	common	UTF-32, UTF-16, UTF-8, ASCII, ISO-8859-1
Slovak	common	UTF-32, UTF-16, UTF-8, ASCII, ISO-8859-1
Russian	ISO 9	UTF-32, UTF-16, UTF-8
	DIN 1460	UTF-32, UTF-16, UTF-8
Slovenian	common	UTF-32, UTF-16, UTF-8, ASCII, ISO-8859-1
Ukrainian	ISO 9	UTF-32, UTF-16, UTF-8
	DIN 1460	UTF-32, UTF-16, UTF-8

Only the supported languages and encodings can be identified. If the input is in an unsupported language or encoding no error is indicated. Sprachinspektor will determine the most similar language and encoding and return this as a result.

The used byte-order of any UTF-16 and UTF-32 input is determined as well. In detail, these encodings are determined as either "UTF-16BE", "UTF-16LE", "UTF-32BE" or "UTF-32LE".

3. Installation

3.1. Requirements

Sprachinspektor SDK merely requires the system's standard C runtime environment.

3.2. What Will Be Installed

The Sprachinspektor SDK contains a dynamic library (DLL/SO), its header file, the code of an example application and this manual.

The Software Development Kit for Linux contains the following files:

```
./doc:
example.c   LICENSE.txt   manual-sdk-eng.pdf

./include:
si.h

./lib:
libsi.so@   libsi.so.1@   libsi.so.1.0.0
```

3.3. Installing the Software

Sprachinspektor SDK is provided as a compressed archive, either in "Zip" or "tar.gz" form, depending on the target platform.

To install the software, just unpack the archive to a directory of your choice and add the library and header files to your project.

3.4. Deinstalling the Software

To deinstall the software, just remove the directory you unpacked Sprachinspektor SDK to.

4. Application Programming Interface

The Sprachinspektor C/C++ library provides an API that is intuitive to use and allows integration into applications easily. All functions and data structures are prefixed "*si_*" to avoid confusions and collisions with other third party library functions and are defined in the header file *si.h*.

Sprachinspektor provides four main functions that determine language and character encoding of a variety of input sources.

→ *si_ffile()* – use a file as input source

→ *si_fstr()* – use a character string as input source (`const char *`)

→ *si_fwstr()* – use a wide-character string as input source (`const wchar_t *`)

→ *si_fnstr()* – use a byte string as input source (`const char *`)

Although they use different sources as an input, all of the above functions return the same data structure, a pointer to a `si_t` structure. This structure contains the determined values for language, ISO 639-3 code and character encoding (see chapter 4.2.1, page 11).

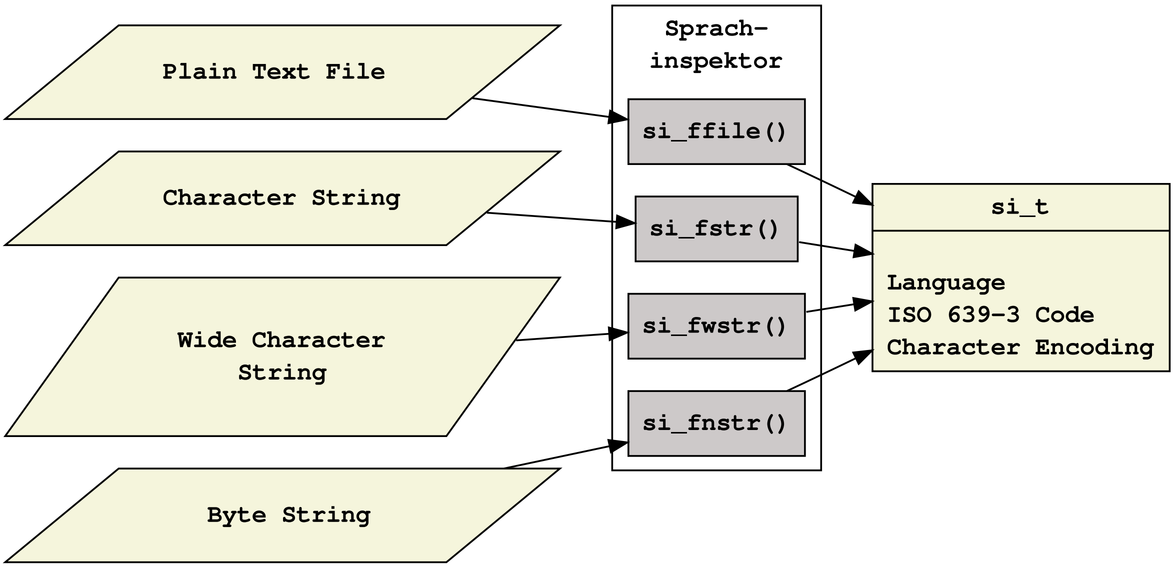


Figure 1: Flowchart of the main Sprachinspektor functions

To assure reliable identification results, at least different characters should be used as an input an provide some degree of variance.

Sprachinspektor does not handle any markup (like HTML or Postscript) and expects every input to be given in plain text format. Documents that contain markup have thus to be preprocessed before they could be used as an input.

If the determined results are no longer needed, you should utilize the *si_free()* function to free all memory used by the result's data structure and minimize the amount of RAM your application allocates.

Whenever an error occurs, Sprachinspektor stores a distinct error code in the pseudo-variable *si_errno* that discriminates the error. Passing this variable to *si_strerror()* reveals the natural language error message associated with the error (see chapter 4.3.6, page 14 and chapter 5.2, page 16).

All functions provided by the Sprachinspektor library are thread-safe and can therefore be used by more than one thread simultaneously.

4.1. A Minimal Application

The following application gives a first overview on the usage of the Sprachinspektor library. Every provided function and the si_t data structure is described in depth in the subsequent chapters.

```
#include <stdio.h>
#include <si.h>

int main (int argc, char *argv[])
{
    si_t        *res = NULL;
    const char *str = "Ein sehr kurzer deutscher Satz.";
                /* Translation: "A very short German sentence." */

    if ((res = si_fstr(str)) == NULL)
    {
        fprintf(stderr, "error: %s\n", si_strerror(si_errno));
        return 1;
    }

    printf("%s, %s, %s\n",
        res->language, res->isocode, res->encoding);

    si_free(res);

    return 0;
}
```

The application uses the *si_fstr()* function to determine the language and encoding of a short, German input string and prints the results. If an error occurs, the application prints the associated error message instead and aborts execution.

```
debian$ ./si-mini
German, deu, ASCII
```

4.2. Important Data Structures

4.2.1. si_t

All main functions of the Sprachinspektor library, those functions which determine language and encoding of an input, return a pointer to a si_t data structure as a result.

All members of the si_t structure are of type char * and can be handled as usual.

Member	Type	Description	Example
language	char *	language name*	"German"
isocode	char *	ISO 639-3 language code	"deu"
encoding	char *	name of the character encoding	"UTF-8"

Figure 2: si_t Members

* The results are represented in ASCII-characters. Whenever a proper language name contains characters that are not encodable in ASCII, the language's name is given in a transliterated form, i.e. *Bokmaal* instead of *Bokmål*.

The structure is formally defined as follows:

```
typedef struct si {
  char *language;
  char *encoding;
  char *isocode;
} si_t;
```

4.3. Function Reference

All of Sprachinspektor's functions and data structures are defined within the header file *si.h*. The header has to be included in all applications that make use of the following functions.

4.3.1. si_ffile()

```
si_t * si_ffile (const char *file);
```

The function takes a pointer to a filename (`const char *`) as an argument and returns a pointer to a `si_t` structure (see chapter 4.2.1, page 11), that contains the determined language, its ISO 639-3 code and the character encoding.

If an error occurs, the function returns a pointer to `NULL` and sets the pseudo-variable `si_errno` to an appropriate value that indicates the error (see chapter 5, page 15).

The file can be encoded in any of the supported character encodings (see chapter 2, page 6). UTF-16 and UTF-32 input is handled.

In order to identify the language and encoding correctly, the file should contain at least 25 characters in distinct words.

As soon as the results are not needed any longer, you should free the memory allocated by the *si_t* structure using *si_free()*.

Any call of the function resets the value stored to `si_errno`.

4.3.2. si_fstr()

```
si_t * si_fstr (const char *str);
```

The function takes a pointer to a character string (`const char *`) as an argument and returns a pointer to a `si_t` structure (see chapter 4.2.1, page 11), that contains the determined language, its ISO 639-3 code and the character encoding.

If an error occurs, the function returns a pointer to `NULL` and sets the pseudo-variable `si_errno` to an appropriate value that indicates the error (see chapter 5, page 15).

The character string may be encoded in any supported character encoding, *except for* UTF-16 and UTF-32. Use *si_fnstr()* for UTF-16 and UTF-32 encoded strings (see chapter 4.3.4, page 13).

> Whenever a string may be encoded in UTF-16 or UTF-32, use *si_fnstr()* instead of *si_fstr()*.

In order to determine the language and encoding correctly, the string should contain at least 25 characters in distinct words.

As soon as the results are not needed any longer, you should free the memory allocated by the *si_t* structure using *si_free()*.

Any call of the function resets the value stored to `si_errno`.

4.3.3. si_fwstr()

```
si_t * si_fwstr (const wchar_t *wstr);
```

The function takes a pointer to a wide-character string (`const wchar_t *`) as an argument and returns a pointer to a `si_t` structure (see chapter 4.2.1, page 11), that contains the determined language, its ISO 639-3 code and the character encoding.

If an error occurs, the function returns a pointer to `NULL` and sets the pseudo-variable `si_errno` to an appropriate value that indicates the error (see chapter 5, page 15).

The character encoding that is internally used for the `wchar_t` data structure is returned.

In order to determine the language and encoding correctly, the string should contain at least 25 characters (not bytes) in distinct words.

As soon as the results are not needed any longer, you should free the memory allocated by the *si_t* structure using *si_free()*.

Any call of the function resets the value stored to `si_errno`.

4.3.4. si_fnstr()

```
si_t * si_fnstr (const char *bstr,
                 size_t      len);
```

The function takes a pointer to a byte string (`const char *`) along with its length (`size_t`) as an argument and returns a pointer to a `si_t` structure (see chapter 4.2.1, page 11), that contains the determined language, its ISO 639-3 code and the character encoding.

If an error occurs, the function returns a pointer to `NULL` and sets the pseudo-variable `si_errno` to an appropriate value that indicates the error (see chapter 5, page 15).

The byte sequence may form a string in any of the supported character encodings (see chapter 2, page 6), which includes UTF-16 and UTF-32.

As *si_fnstr()* handles byte strings appropriate, even if they contain "NUL" characters (ASCII `0x00`), this function should be chosen instead of *si_fstr()* whenever the length of the input is already known.

In order to correctly determine the language and encoding, the sequence should encode at least 25 characters in distinct words.

The parameter `len` has to be set to a value that is lower or equal to the length of the byte string. Due to the technical properties of a byte string, *si_fnstr()* cannot determine the correct length on its own. Severe exceptions may occur whenever `len` is set to a value that exceeds the bounds of `str`, so setting this value deserves special care and attention.

As soon as the results are not needed any longer, you should free the memory allocated by the *si_t* structure using *si_free()*.

Any call of the function resets the value stored to `si_errno`.

4.3.5. si_free()

```
void si_free (si_t *res);
```

The function takes a pointer to a `si_t` structure as returned by *si_ffile()*, *si_fstr()*, *si_fwstr()* and *si_fnstr()* as an argument.

Like the *free(3)* function provided by the standard C library, *si_free()* has no return value.

The memory allocated by `res` is freed completely and will be available for the application again.

4.3.6. si_strerror()

```
const char * si_strerror (int errno);
```

The function takes an error number (`int`) as an argument and returns a pointer to a read-only string (`const char *`) containing the natural language error message.

If an error occurs, you should pass the value of the pseudo-variable `si_errno` to this function in order to obtain the natural language error message associated with the error. A detailed explanation on error handling, error messages and predefined named constants can be found in chapter 5.2 on page 16.

The returned pointer does not have to be and must not be freed using *free(3)*.

4.3.7. si_version()

```
int si_version ();
```

The function does not take an argument and returns a numeric representation of Sprachinspektor's version.

4.3.8. si_version_string()

```
const char * si_version_string();
```

The function does not take an argument and returns a pointer to a read-only string containing Sprachinspektor's version (`const char *`), for example "4.0.0".

The memory pointed to by the returned pointer must not be freed.

5. Error Handling

In case an error occurs within one of the main functions of Sprachinspektor, a pointer to `NULL` is returned and `si_errno` is set to an appropriate value indicating the occurred error ($\neq$ `SI_OK`).

The general error handling policy should be implemented as follows:

1. Return value does not equal `NULL`? → No error
2. Return value equals `NULL`? → An error occurred
   a) Evaluate `si_errno`, handle the error and where applicable
   b) utilize *si_strerror()* to obtain a natural language error message describing the occurred error

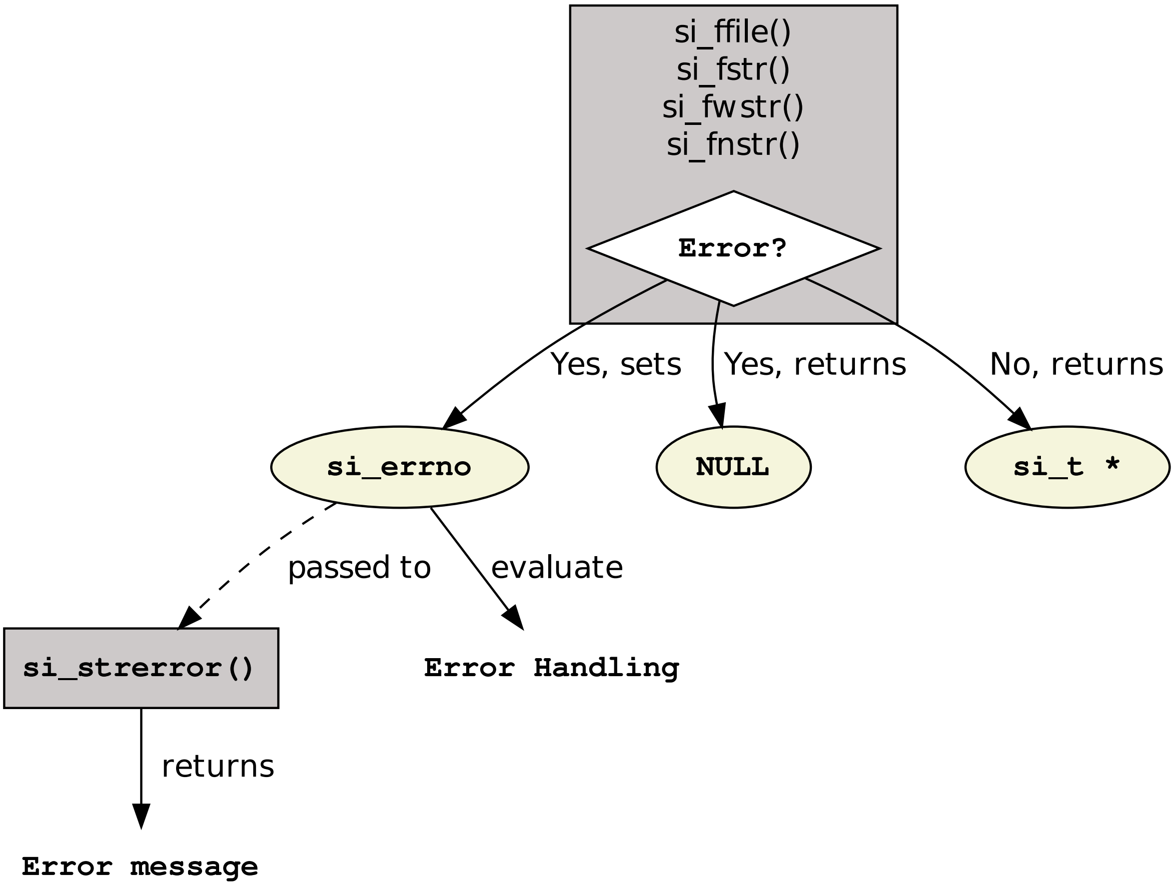


Figure 3: Flowchart of Sprachinspektor error handling

Sprachinspektor's error handling takes allocation of memory into account and frees all allocated memory in every known error path.

5.1. Pseudo-Variable si_errno

si_errno may be used by many threads simultaneously, because it is not implemented as a global variable. The memory necessary for si_errno is allocated on a per-thread basis using Thread-Local Storage (TLS). This way each thread is able to utilize its own si_errno variable.

Nevertheless si_errno can be used as if it was a common global variable[1].

If an error occurs, si_errno is set to a value that discriminates the error. On any call of one of Sprachinspektor's main functions, the value of si_errno is reset to SI_OK.

5.2. si_errno_t Named Error Constants

Sprachinspektor uses the type si_errno_t to provide named error constants for all error cases.

If an error occurs, si_errno is set to a value of type si_errno_t that indicates the error. This way, case dependent error handling can easily be implemented in any application using Sprachinspektor.

The named error constant may, as well as si_errno, be used to obtain a natural language error message describing the numeric error code (see chapter 4.3.6, page 14).

The following table comprises all named error constants used in Sprachinspektor version 4.0.0, accompanied by the error messages returned if passed to *si_strerror()*.

Constant	Error Message
SI_OK	No error
SI_ENOMEM	Failed to allocate memory
SI_EARG	Invalid argument
SI_ESHORT	Insufficient input length
SI_EFOPEN	Failed to open file
SI_EFCLOSE	Failed to close file
SI_EFIO	File input/output error
SI_EMATH	Math error
SI_EUINV	Invalid UTF sequence
SI_EUENC	UTF encoding failed
SI_EUDEC	UTF decoding failed
SI_EBINARY	Binary input data
SI_EUNDEF	Undefined error

Figure 4: si_errno_t Named Constants

[1] Each occurrence of si_errno is replaced with a call to the function *si_errno_location()*, which returns the address of the thread-local variable, by the C preprocessor. As a result, si_errno can be used as if it was a global variable, although it is not. Therefore we call it a pseudo-variable.

6. Hints on Application Development

6.1. Determining Sprachinspektor's Version

After including the *si.h* header, the macro `SI_VERSION_STRING` is available and replaced by a character string containing Sprachinspektor's version by the C preprocessor *at compile time*.

To determine Sprachinspektor's version at runtime, use *si_version_string()* (see chapter 4.3.8, page 14).

A. Example Application

The following example code shows a minimal, but complete application which utilizes the Sprachinspektor library function *si_ffile()* to determine language, ISO 639-3 code and encoding of a set of files that are given on the command line. Errors are handled appropriately by reporting the error to the user and terminating the execution whenever an error occurs.

```c
#include <stdio.h>                                      /* example.c */
#include <si.h>

int main (int argc, char *argv[])
{
    si_t *res = NULL;
    int    i    = 0;

    for (i = 1; i < argc; i++)
    {
        res = si_ffile(argv[i]);

        if (res == NULL)
        {
            fprintf(stderr, "%s: %s\n",
                argv[i], si_strerror(si_errno));
            return 1;
        }

        printf("%s: lang=%s, enc=%s, iso=%s\n",
            argv[i], res->language, res->encoding, res->isocode);

        si_free(res);
    }

    return 0;
}
```

The following output shows an example execution of the application:

```
$ ./example /tmp/english.txt /tmp/german.txt /dev/null
/tmp/english.txt: lang=English, enc=ASCII, iso=eng
/tmp/german.txt: lang=German, enc=UTF-8, iso=deu
/dev/null: Insufficient input length.
```

B. References

→ Lingua-Systems' Sprachinspektor product website,
http://www.lingua-systems.com/language-detector/

→ ISO 639-3 Standard,
http://www.sil.org/iso639-3/

→ The Unicode Standard,
http://unicode.org/

→ RFC 2781: "UTF-16, an encoding of ISO 10646",
http://www.ietf.org/rfc/rfc2781.txt

→ RFC 2279: "UTF-8, a transformation format of ISO 10646",
http://www.ietf.org/rfc/rfc2279.txt

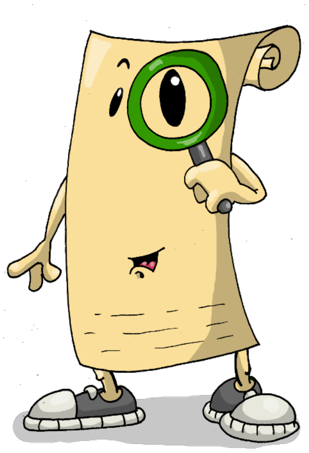
→ ISO 9 Standard (1995) "Transliteration of Cyrillic characters into Latin characters",
http://www.iso.org/iso/iso_catalogue.htm

→ ISO 843 Standard (1997) "Conversion of Greek characters into Latin characters",
http://www.iso.org/iso/iso_catalogue.htm

→ DIN 1460 Standard (1982) "Conversion of cyrillic alphabets of slavic languages",
http://www.nabd.din.de/

→ DIN 31634 Standard (1982) "Conversion of the Greek alphabet",
http://www.nabd.din.de/

→ Streamlined Sytem (1995) "Romanization of Bulgarian",
http://members.multimania.co.uk/rre/Streamlined.html



http://www.lingua-systems.com/language-detector/

Index

D

E

I

M

R

S

T